

# **BSoD/Introduction to Physical Simulation**

**by Felipe Bergamin Boralli**



# Table of Contents

Basics of Math and Physics.....	5
Vectors.....	5
Particle Simulation.....	5
Rigid Body simulation.....	5
Physics of deformable bodies.....	6
Fluid Simulation.....	7
Further information.....	9
Fluid simulation.....	10
The Basics.....	10
Example of a fluid type in a colision with a cube.....	16
Example of a inflow-outflow setup.....	18
Fine tuning of properties.....	19
Soft Body.....	20
The Basics.....	20
Towel falling onto table.....	22
Animating a soft block hitting a wall.....	23
Simulating a flag in a pole.....	25
Fine Tuning of Properties and Hints.....	27
Particles.....	29
Introduction.....	29
Particle Interaction.....	31
Simulating sparkles.....	33
Emitting objects as particles.....	35
Good links to follow.....	36
Rigid Bodies.....	37
The basics.....	37
Dominoes.....	38
Effective use of the game engine.....	40
Huge simulations.....	41
Hints.....	42
Additional information.....	43



# Basics of Math and Physics

These subjects are usually seen by undergraduate students of sciences and engineering. However, a basic understanding of them is required if you want a deep understanding of any computational physical simulation.

The text below is not intended to be a reference in these subjects. It's main purpose is just to provide the user of Blender some fundamental insight on what is actually happening inside physical simulations of Blender. If you wish get further knowledge on any topic, there is also some useful links at the bottom.

## Vectors

The formal description of what vectors are and how to use them (mathematically) is in the core of Linear Algebra. However, the definition of vector in Linear Algebra is somewhat obscure, and not intended for direct geometrical practical use, being defined as an abstract concept.

Being so, in this document will be used the approach used in analytic geometry, being vectors as "arrows" in 3d space. Examples of physical vectors are velocity, forces, and accelerations. Physical quantities that are not vectorial are said to be Scalar. Examples of scalar quantities are energy, mass and time.

Some properties that summarize vectors:

- One vector may be decomposed in it's components on the main axes
- Two vectors are equal if, and only if, all their components are equal.
- The sum/subtraction of two vectors is equal to the sum/subtraction of their components.
- There is no multiplication/division of vectors as there is in "common" numbers.

There are some types of operation called products, but any of them are not equal to the products of the components, as you might think. These will not be discussed here.

## Particle Simulation

Particles are the most basic type of body you can create in Physics. They have no size, and since they have no size they cannot be rotated. The law that governs the movement of the particles is the Newton's 2nd law:  $\mathbf{F} = m\mathbf{a}$  (Forces equals mass times acceleration). This is not the complete form of Newton's law, but the simplified form is ok for particles. The vectorial sum of all the forces acting on the particle equals has the same direction and orientation of the acceleration. The branch of Physics that is dedicated to this subject is called Classical Mechanics.

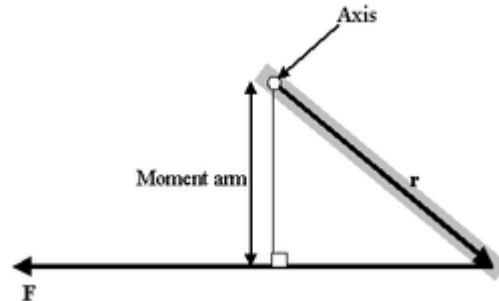
Using vector notation, this equation is simple to solve by using a computer even if you use multiple forces of different types like damp (a force that is proportional to speed, the higher the speed, the higher the force) or the force fields present in Blender. Being simple to solve, Blender is able to calculate the position of thousands of particles almost in real-time when animating.

## Rigid Body simulation

Rigid body means the body is not deformable, i.e. cannot stretch, shrink, etc. The main difference from particle simulation is that now our objects are allowed to rotate, and have a size, and a volume.

The equation that governs rotational motion is  $\mathbf{T} = \mathbf{I} \boldsymbol{\alpha}$ . Torque equals moment of inertia times angular acceleration. Now we need a definition of each of these words are:

A force may cause a torque. The torque it causes is the *vectorial cross product* of the component of the force perpendicular to the axis you are evaluating by the distance from the point of the application of the force to that axis. Also, torques are vectors. A very useful special case, often given as the definition of torque in fields other than physics, is as follows:



### The moment arm diagram

The construction of the "moment arm" is shown in the figure below, along with the vectors  $\mathbf{r}$  and  $\mathbf{F}$  mentioned above. The problem with this definition is that it does not give the direction of the torque but only the magnitude, and hence it is difficult to use in three-dimensional cases. If the force is perpendicular to the displacement vector  $\mathbf{r}$ , the moment arm will be equal to the distance to the centre, and torque will be a maximum for the given force. The equation for the magnitude of a torque arising from a perpendicular force:

For instance, it is much easier to close a door by pushing it by the handle than by pushing it in the middle of the door, because, when you do it by the handle, you increase the distance of the force you are applying to the axis of calculation.

Moment of Inertia is a measure of how difficult is to rotate the body. It is proportional to both the mass and the geometry of the object. Given a fixed volume, a sphere possesses the smallest moment of inertia possible.

Angular acceleration: Is a measure of the acceleration in the rotational movement.

The programs devoted to deliver fast and accurate simulation of rigid body dynamics are often called physics engines, or game engines. Blender itself has a game engine, called Bullet. All simulations in Bullet engine, if well designed, are real-time, except for those with a very high number of objects present.

In the near future, it is expected that computers used for gaming will have a Physics Processor Unit (PPU) dedicated to these calculations as a card, like what happened when graphics processing was moved from the CPU to video cards (Graphics Processing Units or GPUs).

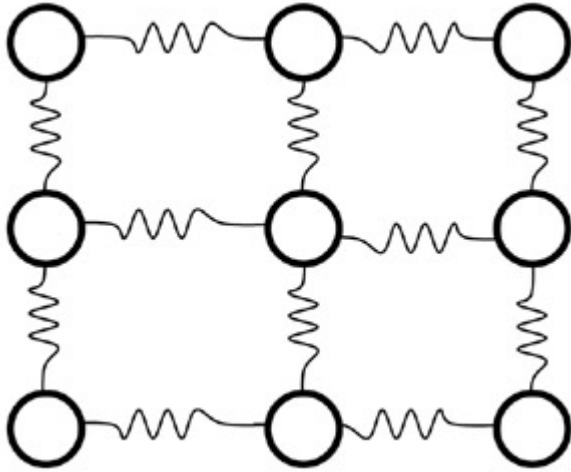
## Physics of deformable bodies

The method most computers use to simulate deformable 2d or 3d bodies is to subdivide (automatically or manually) the body in cells, and then, considering any properties inside a single cell to be an interpolation of the properties on the corners, and we just have to solve the equations on the boundaries of the cells. If you use a great number of cells, you will get results very similar to the reality.

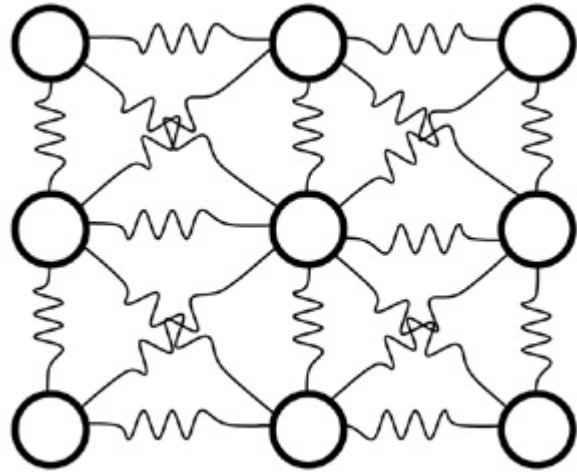
A common application of this method, plus some more hypotheses, leaves us into a wide field of

engineering which nowadays is called Finite Element Analysis.

In blender, the simulation of deformable bodies is somewhat similar to this, but with an approximation. If we enable an object to be a soft body, Blender considers that all faces are cells where vertices are masses, and the edges are springs. With these pictures, you can understand how a 3x3 vertices plane is simulated in Blender.



**A cell in Blender**



**A cell in Blender, with Stiff Quads enabled**

## Fluid Simulation



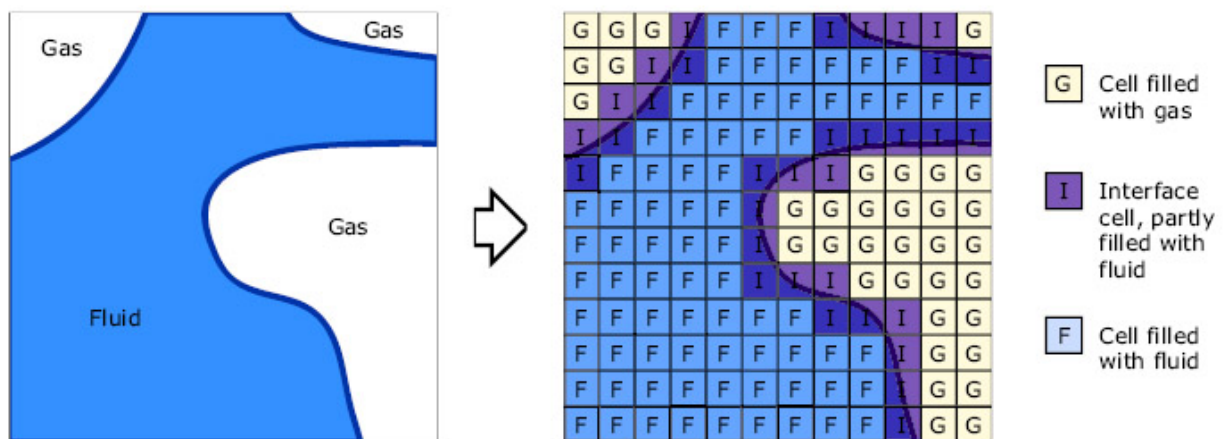
Turbulent flow and laminar flow are visible in the surface of water around the submarine. In the nose of the submarine, the flow is laminar (without bubbles, organized). After the nose, the flow modifies itself into turbulence)Turbulent flow is defined as: "Apparently random flow that is not random but defies our ability to analyze it at this time." We cannot predict for sure how will the

velocity of each point, only the mean velocity of the fluid. For most applications using fluids, they are using turbulent flow. When the flow of a fluid is not turbulent (you can predict the velocities in any point), the flow is said to be laminar. Most of the flows you are able to see in your daily life are turbulent.

There is a theory (in the scientific use of the word theory, it has been proven), that provides us a set of equations called Navier-Stokes equations, that completely state how a fluid will behave in most of situations being turbulent or not. However, these equations cannot be solved by hand, and today, no exact answer has been found to solve this equation in its complete state, just in some very special states, and none of them include turbulence.

Currently, the method used to solve this equation consists in using iterative solving, thus getting an answer as close to real as we want to. This is called DNS (Direct Numerical Simulation). However, DNS requires enormous computation power, and even for today, mid-2006, only supercomputers or very large clusters of computers can use DNS with some success. If correctly applied, however, DNS returns the best and most trusted results of all methods, from the more micro distance at which the results are meaningful to the simulation, to the more macro distance, the scale of the objects we are simulating.

So, we need to approximate our model more, in order to do less calculation. Instead of considering the fluid a continuum, we will discretize our fluid. By discretizing, you can understand dividing into cells. Inside a cell, the properties like velocity, pressure, density are all considered to be the same, so we only have to solve equations on its borders. We also discretized time, i.e., only some instances are calculated. However, we need another equation that deals with this discrete problem. This equation is called the Lattice-Boltzmann equation, and this equation complies with the Navier-Stokes equation.



### How the fluid is divided in cells

There is also one more optimization done in Blender, the use of adaptive grids. In a region far from interfaces, instead of using a tiny cell, we use a larger cell. This greatly decreases calculation time (up to 4 times faster), without loss of quality. This optimization is responsible to find places where you can use a larger cell without disturbing the results, and when to start using smaller cells in these places.

**Boundaries:** The boundaries (domain, obstacles) counts as a cell in the method.

- **No slip:** The fluid cells near the surface of the boundaries are not allowed to move at all, having zero velocity.



- **Free slip:** The fluid cells are allowed to move freely. If the calculation indicates that that cell would move inwards the boundary cell, then its velocity vector is inversed.

## Further information

### Vectors

- Wikipedia: [\[1\]](#)
- University of Ghelph: [\[2\]](#)

### Particle simulation

- Classical Mechanics: [\[3\]](#)

### Rigid Body simulation

- Torque: [\[4\]](#)
- Definition of Physics engine: [\[5\]](#)

### Finite element analysis

- Finite Element Analysis : [\[6\]](#)
- Finite Element Analysis example: [\[7\]](#)
- OpenSource finite element analyzers
  - Impact: [\[8\]](#)
  - Elmer: [\[9\]](#)

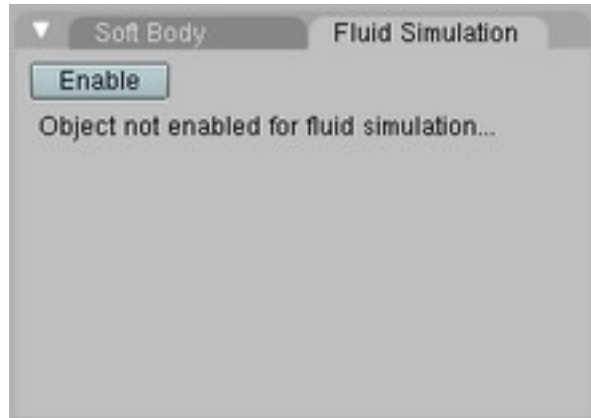
### Fluid Simulation

- A scientific paper describing the method - : [\[10\]](#)
- A implementation in Blender: [\[11\]](#)

# Fluid simulation

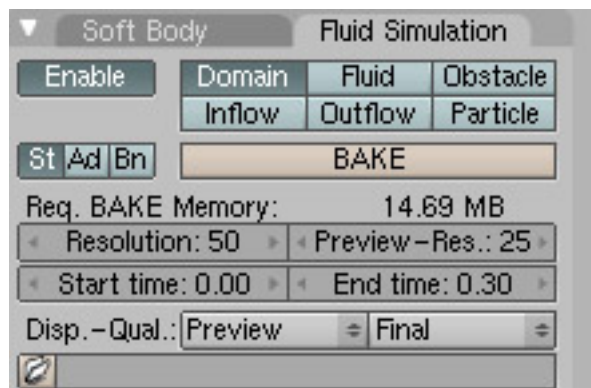
## The Basics

So, when you get to the fluid simulation area, initially your object is not enabled for fluid simulation.



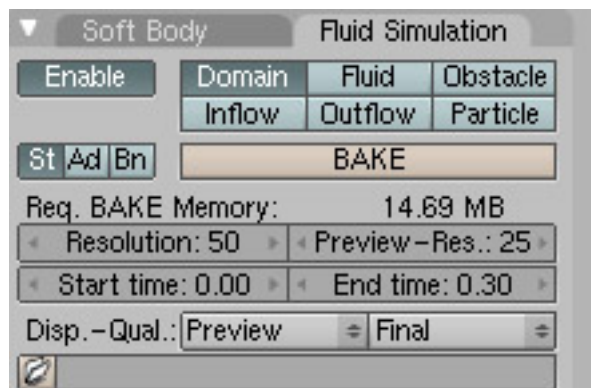
## Domain Type

Enabling it will flag the object so when we call the simulation module, it will “look” for all flagged objects to begin the simulation.



## Domain Type

First, for a simulation to be done, we need the domain. The domain of the simulation is a box where the fluid calculation will be done. Try using a cube for the domain, if you use an irregular sized or shape object, only its bounding box will be used, (the length of the sides can be different)



## Standard Properties of the domain type

## Domain:

### Standard (St)

- **Resolution:** this is a very important property to be selected. It will determine the “detail” of the simulation. Better explained: as we have seen, the calculation is made in cells, each cell having the same properties for the fluid. A resolution of 50 means you will have  $50 \times 50 \times 50 = 125,000$  cells for calculation. If the domain object has not all sides of equal length, the longest side will be used, so each cell remains cubic. Higher resolutions improve the detail of the simulation, but also increase the memory used (both RAM and HD) and baking time. You only see this mesh when rendering.

Memory use increases dramatically as you increase resolution:

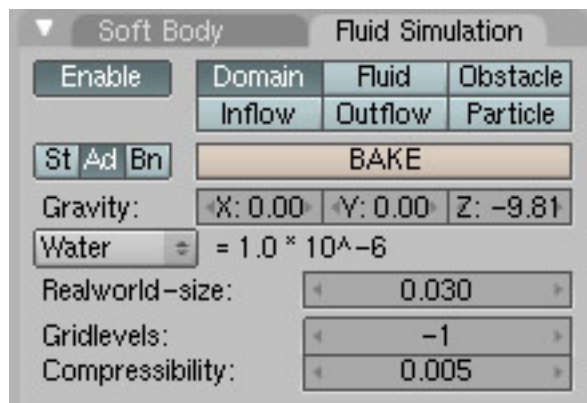
Resolution	RAM Memory (Megabytes)
32	4
64	30
128	240
256	1900
512	15000

- **Preview-Res:** (Preview Resolution) the same as Resolution, but is used on the preview (**Alt+A**), and does not influence calculation time (as long it is less than the Resolution). Limited to 100. It's purpose is that a very large real-time simulation would probably freeze Blender. As it is only an approximation, what you see in preview and what you get in Render are not always the same.
- **Start – End times:** The time in seconds that you want to be baked.

If you set the start time higher than zero .....

This time will be distributed among the frames selected for animation in Render|Animation (F10 Scene Buttons)

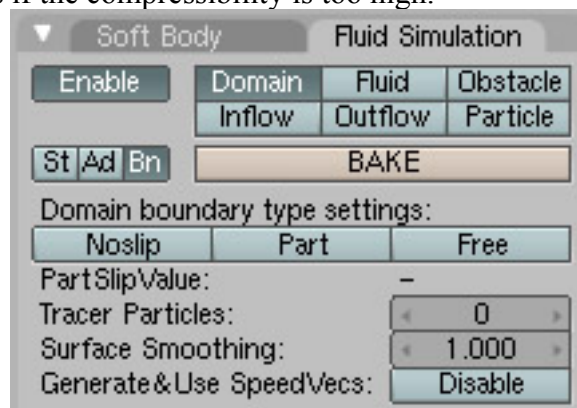
- **Disp. –Qual.:** Select what resulting meshes will be displayed in rendering time and in development time.



### Advanced Properties of the domain type

## Advanced (Ad)

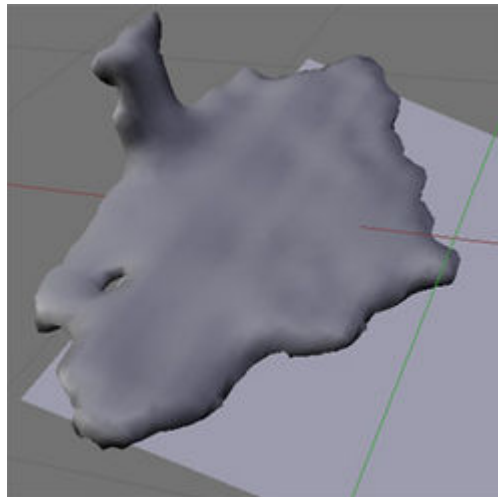
- **Gravity:** the acceleration on all the fluid particles. Measured in  $\text{m/s}^2$ .
  - The fluidsim currently does not support zero gravity in the z axis. Use a very small negative value for gravity if you want to achieve a non-gravity fluidsim.
- **Viscosity** (the combo box with water selected): Selects the viscosity of the fluid. Currently, it is not possible to work with two fluids of different viscosities. All the fluid inside the domain will have the same viscosity.
- **Realworld-size:** The size of the longest side of the domain, in meters. It will greatly influence the results of the simulation. Depending on Realworld-size, you may have a drop falling on a cup of water, or some tonnes of water falling into a lake.
- **Gridlevels:** The number of coarsened (with less cells) grids that will be done, for approximations to speed up calculation. Leave at -1 for auto. 3 coarsened grids may cause the simulation to run up to 4 times faster.
- **Compressibility:** For the calculation of the simulation to be feasible, we need to consider the compressibility of water. Increasing this value may decrease simulation time a bit, but it is not recommended, as it will decrease the realism. Also, large portions of fluid standing may have problems if the compressibility is too high.



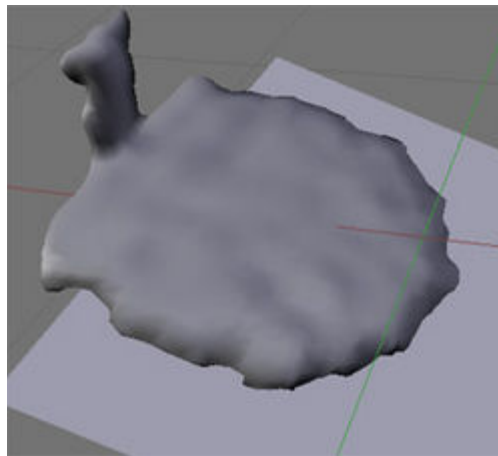
## Boundary Properties of the domain type

### Domain Boundary (Bn)

- **Boundary Type Settings:** (also valid for obstacles)
  - **Noslip** - The fluid cell in contact with the domain boundary is not allowed to move.
  - **Part** - The fluid in contact with the domain boundary receives an attrition force, but is allowed to move. This is defined in PartSlipValue.
  - **Slip** - The fluid in contact with the domain boundary may move freely.

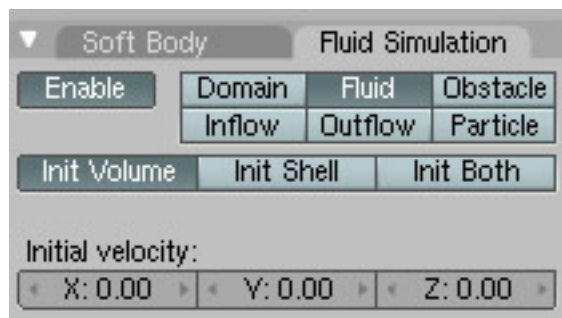


### FreeSlip Simulation



### NoSlip Simulation

- **Tracer Particles:** The number of tracer particles that will be put into the fluid at the beginning of the simulation. To display them you will need to create a particle fluid type, which will be discussed later.
- **Surface Smoothing:** Amount of smoothing to be applied to the fluid surface. 1.0 is standard, 0 is off, while larger values increase the amount of smoothing.
- **Generate&Use SpeedVecs:** When this button is marked as disabled, no speed vectors will be exported. By default, speed vectors are generated and stored on disk. They can be used to compute image based motion blur with the renderoutput nodes.



### Fluid type

## Fluid

### Init Volume



Example of the different volume init types: volume, shell and both. Note that the shell is usually slightly larger than the inner volume.

- **Volume Init Type:**

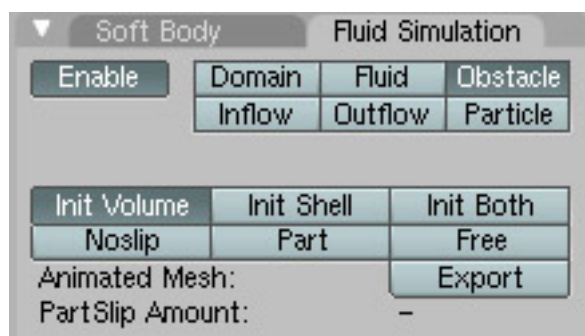
**Volume Init** will initialize the inner part of the object as fluid, this only works for closed objects.

**Init Shell** will only initialize a thin layer for all faces of the mesh, this also works for non closed meshes.

**Init Both** combines volume and shell, the mesh also should be closed. Also see the picture above. As you can see, the only way to use a plane or any non-closed mesh as an obstacle is to set it to Init Shell. If you don't, you will get bad results.

**Attention:** Volume Init Type is used on other types of objects inside fluid simulation, but the meaning is the same as above.

## Obstacles

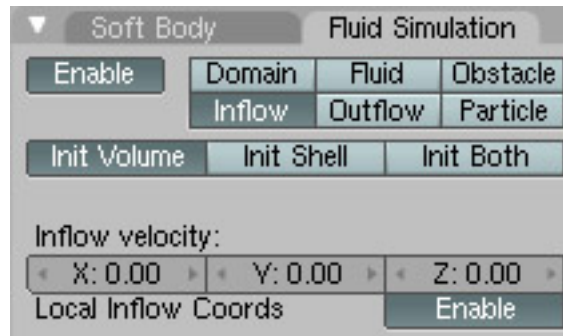


### Obstacle Type

Obstacles are as the name implies, i.e. an object that is placed in the fluid simulation to obstruct the flow. You can only define a mesh to be an obstacle. It can be animated, and the calculation of the fluid will respond accordingly.

- **Animated Mesh:** If the mesh itself is animated (bones, shapekeys, etc) check this box for it to be computed. The computation will be slower, so only activate it when needed. If only the object is animated, but the mesh remains the same, do not activate this option.

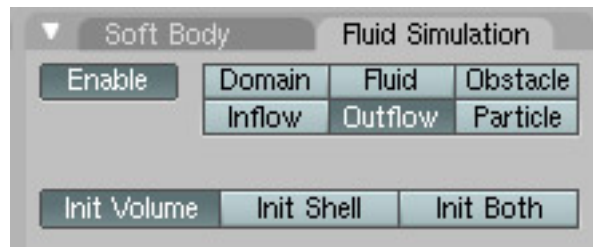
## Inflow



### Inflow Type

Inflows will inject water inside the domain. The amount of water put inside the domain is defined by both the area of the cross section perpendicular to the flow and the velocity set. Take care to not fill up the entire domain, or the calculations will be severely downgraded.

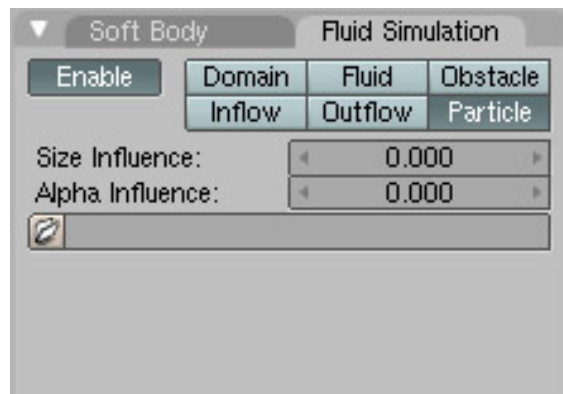
## Outflow



### Outflow Type

Outflows will take fluid out the domain. The amount of water put inside the domain is defined by both the area of the cross section perpendicular to the flow and the velocity set.

## Particle



### Particle Type

After clicking on particle, an additional side panel (Particle Panel) will be created. The type of particles created are tracers, i.e., they will follow the fluid stream during the simulation. You do not need to worry about it, the particles will be created in the correct position(s). If you happen to move the object, you will need to delete the particles and create them again. Also, most of the settings, such as Fields and Deflection don't work on Fluid Particles.

- **Size influence:** The particles may have different sizes. If zero, all will have the same value. The higher the value, the higher the difference.

- **Alpha influence:** When a nonzero value is input, bigger particles will be transparent, and smaller ones will be opaque. The higher the value, the higher the difference.
- **Bake directory:** The directory where particle simulation data will be written. For most cases, the same directory is used for both the particle and fluid simulation data.

## Side Notes

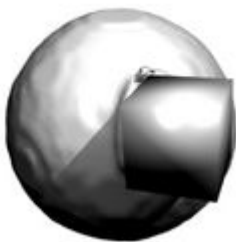
The simulation generates two meshes, one called Preview, and other Render.

After you bake, the domain box turns into the simulated fluid. The fluids, inflows, outflows stay intact. If you no longer wish to rebake your simulation, you may delete them if you wish

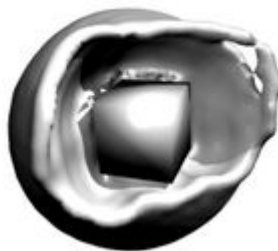
When you wish to send a baked simulation to someone, you will need to send the .blend file, also all files that are written to the fluidsim / particle directories.

## Example of a fluid type in a colision with a cube

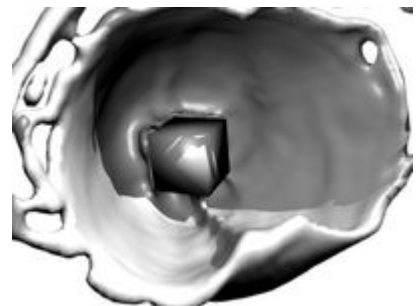
This example aims to show many abilities of the fluidsimulation engine. We will simulate the collision of a blob of fluid with a cube. If you follow my notes accordingly, you will be able to do like these stills.



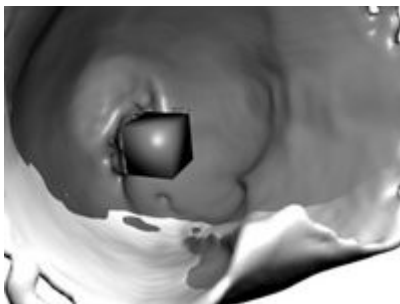
frame 6



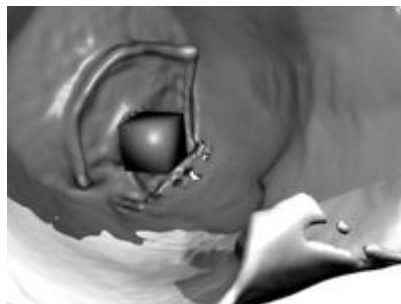
frame 10



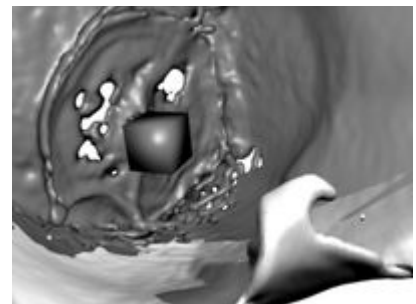
frame 15



frame 20



frame 25

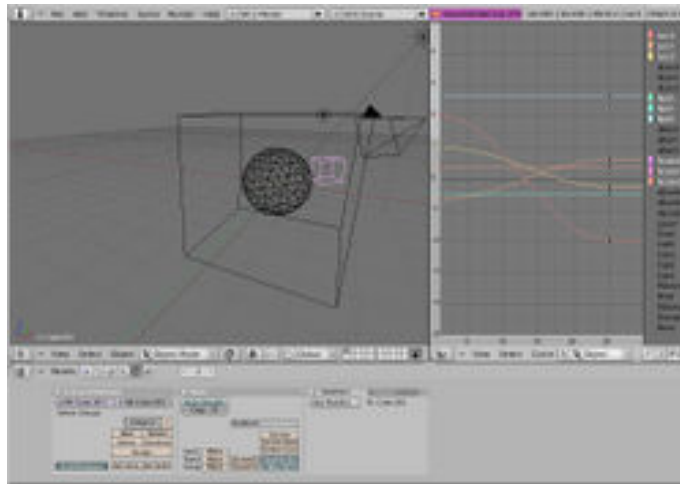


frame 30

## How to:

To be able to do this, you need to know the basics about the Blender interface, how to create and move simple meshes, and how to do keyframe animation with an object.

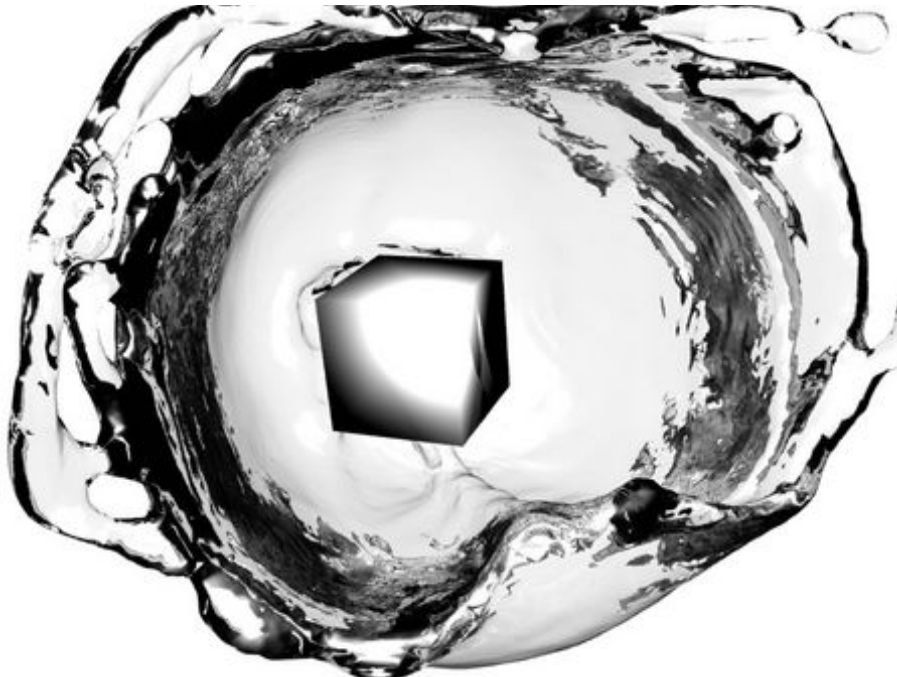




The setup before baking

1. Select a cube, set it as a domain, with a low gravity (gravity Z = -0.1)
2. Set the directory you wish to save the baked fluid data files in
3. Create an isosphere inside the domain, set it as a fluid type
4. Create another cube inside the domain cube, set it as an obstacle.
5. Go to render settings (F10) and change the end of the animation to a smaller value, say 30.
6. Animate the cube so that it will cross the the blob of fluid
  1. To do this, create a keyframe at frame 1 pressing **I**, and select *LocRotSize*
  2. Go to frame 30 by pressing **Up Up Up**, then move the cube to a position where it would have crossed the blob of fluid, and press **I**, and select *LocRotSize*
7. Bake!

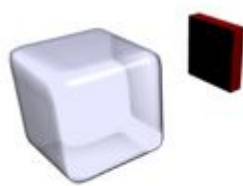
Depending on your processor, it may take some 10 minutes to get it done. Now you have it baked, you can see the results by pressing **Left** or **Right**, to move between the frames. To get better results, set a waterish material in your fluid, and render in YafRay.



YafRay Render. Rendertime = 2 hours on a 2GHz processor, two subdivisions. Except for the cube, one cannot determine if this image is either a photo or a 3d image

## Example of an inflow-outflow setup

This scene will illustrate the use of inflows and outflows. The setup involves a inflow that injects water in the domain, facing the outflow.



frame 1



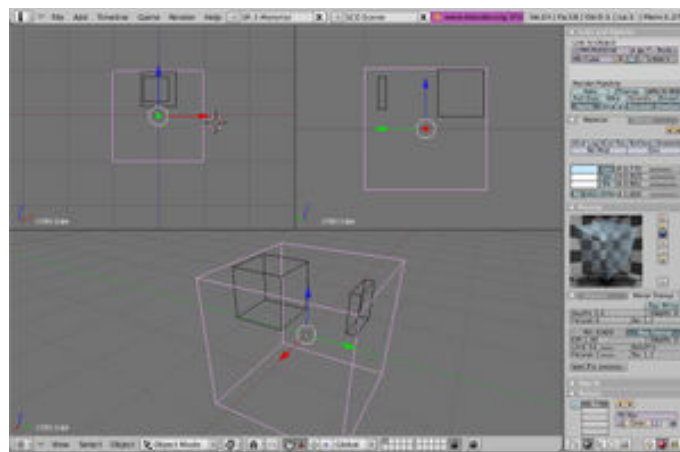
frame 8. The fluid has hit the outflow. Most of the fluid is now exiting the domain from the outflow.



frame 14



frame 24



### The scene setup

#### How to:

To be able to do this, you need to know the basics about the Blender interface, how to create and move simple meshes, and how to render.

1. Create a cube, set it as a **domain**.
2. Set the directory you wish to save the baked fluid.
3. Create a cube inside the domain cube, set it as a **inflow**, and with  $V(y \text{ direction}) = 1.00$
4. Create another cube inside the domain cube, scale it in one axis, set it as an **outflow**, facing

- the direction of the inflow, just the the render setup scene.
5. Go to the render settings (F10) and change the end of the animation to a smaller value, say 30.
  6. Bake!
  7. Create a waterish material (tweak *ZTransp* settings). *Set as smooth* the resulting fluid.
  8. Render the animation.

## Side Notes

If you want to control the volume of inflow or outflow over time you can either scale the inflow/outflow object (which you can animate) or move the inflow/outflow object out of the domain.

## Fine tuning of properties

Things you can do to improve your fluid meshes when rebaking:

- Increasing the resolution
- Keeping you obstacles, inflows, outflows with less polygons as possible
- Avoid using situations when fluid will be at very high velocities. The calculation time will be high, and results may not be accurate

Things you can do to improve your fluid meshes without rebaking:

- Set the fluid as a smooth surface
- Subsurf the fluid once

Things you can do to improve the overall rendering quality:

- Modify the material of the fluid meshes to get a waterish-like aspect. Use *Ztransp* to do that.
- Render using a external Raytracer like YafRay or POVRay to get better refractions inside the fluid and caustics

# Soft Body

## The Basics

### Main Panel



### The Soft Body panel enabled

- **Enable Soft Body:** Enables this object to act as a Soft Body
- **Bake Settings:** Enable the simulation to be saved as vertex positions.
- **Friction:** A generic force against movement that acts on all vertices. A value of zero means no Friction
- **Mass:** The mass of the body in kilograms. Will be shared equally among all vertices. A higher mass will make the object harder to stop, and the action of force fields will be smaller
- **Grav:** The local gravity, it's always pointing the negative z-axis
- **Speed:** A tweak used while solving the movement. Don't modify, unless you have a good reason to do so.
- **Error Limit:** The biggest ammount of error the calculation solver may commit. A smaller value means more realistic results, but more baking time also.
- **Use Goal:** Enables creating a force to try to keep all the vertice to a position. Use the double arrow on the right of Use Goal to select what vertex group you wish for all the vertices to try to keep position.
- **Goal:** If no vertex group is defined, this defines that all vertices should try to keep their respective position.
- **G Stiff:** The intensity of the force that will be onto all vertices (even those not in direct contact with the goal). Most of the times, this is set to zero, or a very small value.
- **G Damp:** This adds some friction to the Goal Movement.
- **G Min and G Max:** These redefine the weight of the vertex groups defined, if any.
- **Use Edges:** Add springs on edges. For most of the times, keep this on.
- **Stiff Quads:** On faces with for vertices, add springs also on the diagonals
- **E Stiff:** The spring stiffness. A value of zero means the spring offers no resistance to movement (It all happens as if no spring was there). A value of one means the spring cannot be stretched (as if was there a solid rod). However, a value of one would make calculations impossible, so, the maximum is 0.999 . A elastic material has E Stiff on the range of 0.3 to 0.7 . Most type of cloths, like cotton, leather, linen, etc ; that cannot be stretched a lot have E

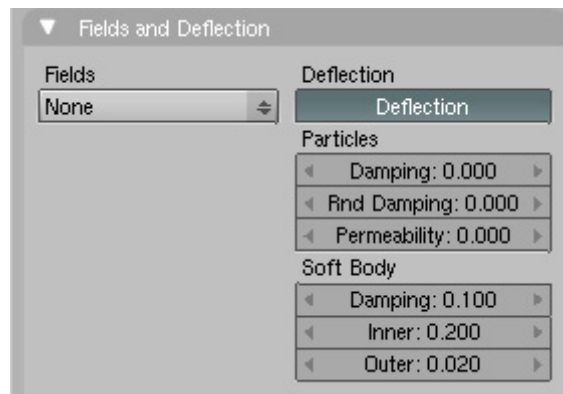
Stiff on the range of 0.95 to 0.99

- **E Damp**: Add some friction to the movement of the springs. If you set this to zero you may get a non-real, never ending movement.

## Bake Settings

- **Start**: The starting frame
- **End**: The last frame
- **Interval**: The interval between frames that a shape key of the object will be saved. Unless this will cost you much memory, keep this at 1, by storing all frames, to achieve more realistic results
- **Bake**: Starts processing the softbody interaction between the frames selectes. After you bake, if you enter in editmode on the softbody object, when you leave, it will erase the baking, even if you haven't modified anything. Try putting the use of Bake at the very bottom of your production flow, one of the last step, mainly when it's a baking that takes some time.

## Deflection Settings



### The deflection panel enabled

Now, we only use the softbody settings:

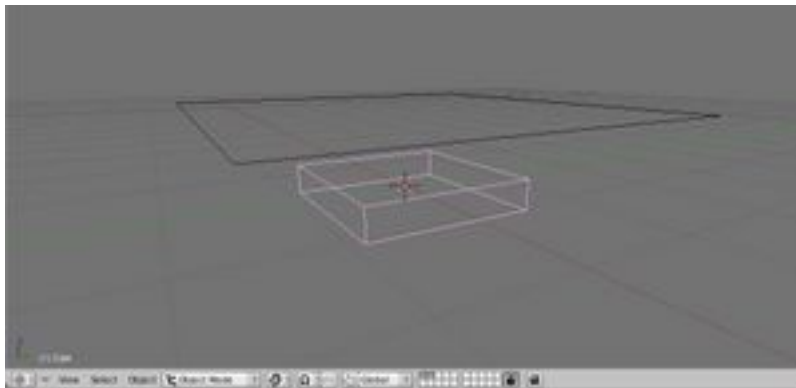
- **Damping**: The some friction that happens upon the softbody contact.
- **Inner**: The maximum lenth the softbody may get inside the object.
- **Outer**: A distance from which the softbody starts execting forces on the object.

## Towel falling onto table

As our first example, we will try to simulate a towel falling onto a table.

**Pre-requisites:** You need to know Blender interface, and the very basics of mesh modelling (scaling, moving)

We will need a cube, to be the top of the table, and a plane, to be the towel.



**The scene setup setup. The plane on top is the towel, the box below is our table.**

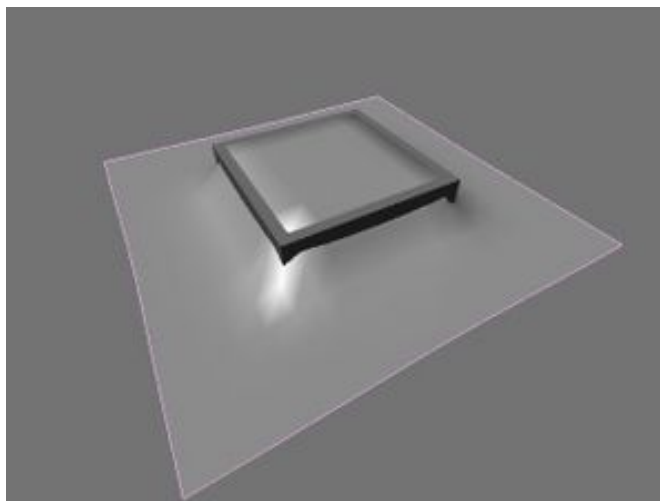
Then, while the plane is selected, enter editmode, enter Editing **F9** select all vertices, click subdivide some times (4 to 6 times), then leave editmode, click **Enable Soft Body**.

**Do not forget to subdivide the plane mesh.** Only the vertices are taken into account in the collision calculation. If you set a collision object, and it doesn't face any vertices, the softbody will pass through.

As we want our towel to fall onto the table, we will need some gravity. Set gravity to 10. The gravity here points to Z negative global axis. Also, deselect **Use Goal** as we want our towel to move freely.

Select our table, and go to the object panel (**F7**), then **Fields and Deflection**, check the box **Deflection**.

If you bake the softbody now, you will already get some good looking results. Problems may arise if the borders of the towel touch each other, they may keep fixed. Try to avoid this, either by decreasing **Error Limit** or by scaling down the towel, so it would be physically impossible for it to happen.



The first result after baking. You can see the towel has entered inside the table. Now we will fix it by increasing Outer.

After this first result, the towel was rotated 45 degrees, the parameter Outer was increased, and a nice texture was applied. The render results.



frame 15



frame 20

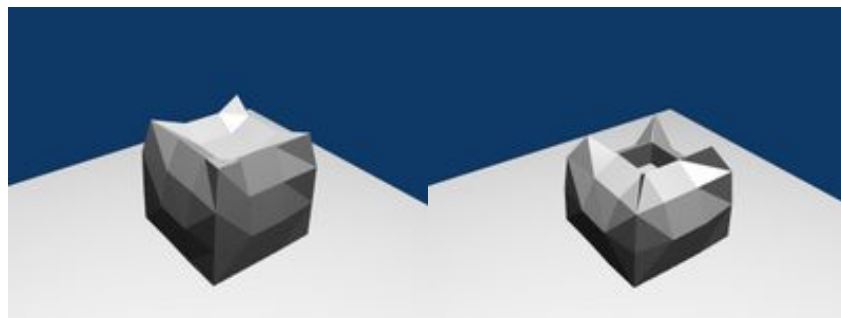


frame 30

## Animating a soft block hitting a wall

What we call here as block is a 3d object with an internal volume, like a cube, a cone, or a cylinder with the ends closed. A plane, or a cylinder with the ends open have no internal volume.

As we have seen, the method Blender uses to simulate a softbody is to put masses in the vertices, and springs on the edges. If you use the same method you used to simulate the towel to simulate a cube hitting a wall, (Add a cube, subdivide it some, and simulate it's collision with a solid plane, it will probably collapse (fall inside itself, like the rendered picture below). So why does this happen?



Frame 14. Impact

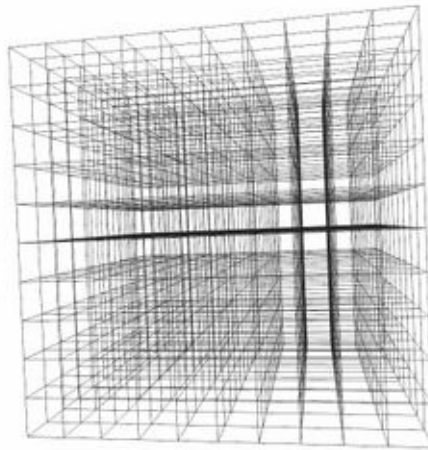
Frame 19



Frame 24

Frame 35. Fully collapsed structure.

It happens because with the method used by Blender, this cube simulated is like a cube of a thin foil (like aluminum foil) falling. Of course, it is unavoidable for a cube of thin foil to be a part of such a event and be intact after.



### The type of block needed, with vertices inside it

So to simulate a good collision of a volumetric solid, we need to build a mesh that also contains vertices **inside** it, in an organized way, so the object resists to be compressed or stretched also inside, not just the walls. The approach we will use consists in using small cubes as build blocks of our objects. It can be seen from the images below that it produced good-looking results.

*This is one method to build such a mesh. You can use others if you want.* Insert a cube, scale it to half the size (exact). Enter editmode, enter Editing **F9** select all vertices, **Step** = 6 (in our example) and while in front view click **Extrude Dup**. Don't leave edit mode. Go to top view, select all vertices, and **Extrude Dup**. Go to side view, select all vertices, and **Extrude Dup**. If you done everything correctly, you will arise to a mesh like this. Now, select all vertices, and click **Remove Doubles**, to remove all overlapping vertices that arisen from extrude dup. Check ok.

#### Attention for normals

In the softbody solid example, when we remove the doubles, half the face normals are up, and half will be down. If you will use something that depends on the face normals, like Set as Smooth, you will need to manually flip the normals in the correct way, since the command Recalculate Normals Outside does not work.

Now go to the object panel (**F7**). Select the plane below the cube, and enable it for softbody deflection. Select the cube, and **Enable Soft Body**. Select a **Gravity** of 10, uncheck **Use Goal**, use **Stiff Quads** select **E Stiff** to about 0.4. Use the **Bake Settings**, and bake. It may take a while, about 2 to 5 minutes to bake 100 frames.

The results. Even being simple to do, the final results are very realistic. A much more realistic approach would be possible if you did a mesh with more cells. It was 6 on our example. However, if you use a side of 12 cells, you will increase calculation time 8 times. If you use a side of 25 cells, the calculation time will be about 70 times our example.





Frame 29. Impact



Frame 36. Compression



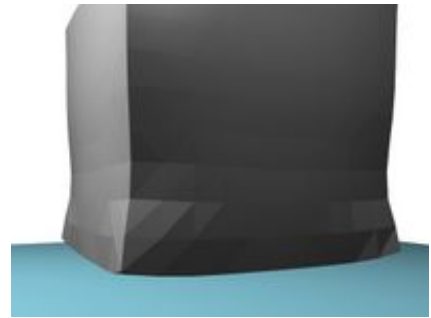
Frame 40. Max Compression



Frame 51. Started bouncing back



Frame 52. In the air again

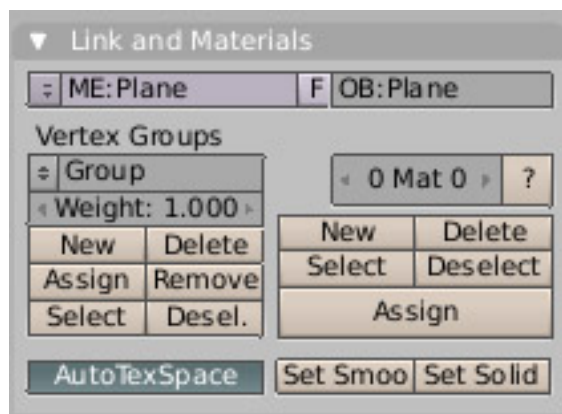


Frame 56. In the air. Stretching.  
Very realistic "jelly" effect

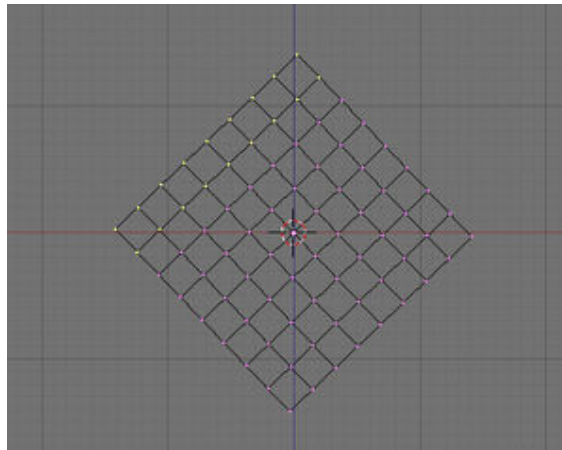
## Simulating a flag in a pole

With this example it will be seen how to use forces on softbodies. We will simulate a flag on a pole with wind.

Add a plane in the front view, rotate it about 45 degrees on the y-axis. Enter editmode, subdivide the mesh three times, and we now will select which vertices will be fixed by creating a vertexgroup. Select the vertices as shown in the picture, then create a new vertexgroup with them selected (we will call it "pole").

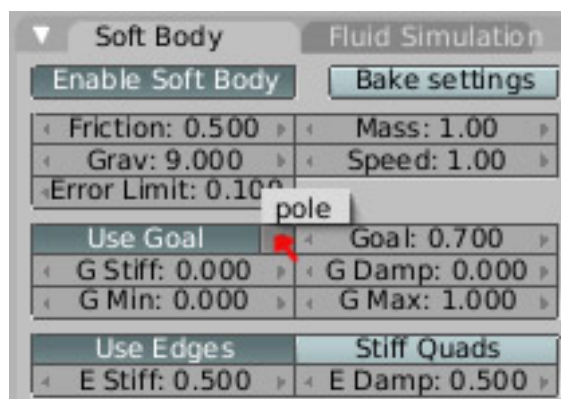


Assigning the vertex group



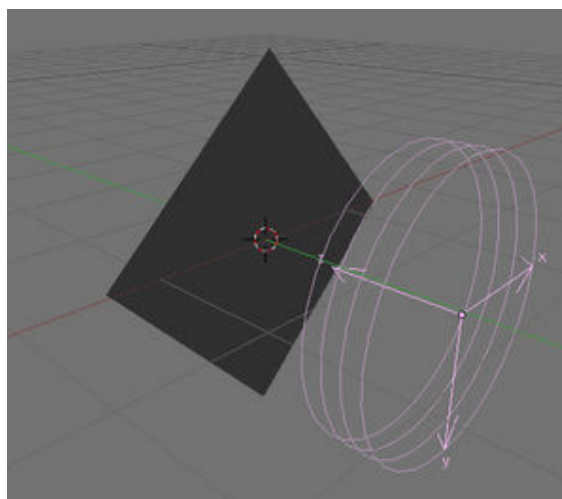
**The vertex group selected: The yellow dots**

Now, enable the flag to be a Soft Body, and use the vertices we selected as Goal.



**Enabling the selected vertex group to be the goal**

Select **G Stiff** as zero, and **E Stiff** as 0.975. If you see what is happening to the flag by either baking or advancing frames, you will see that it only falls down. It happens because we don't have wind yet. To add wind, press **SpaceBar** > **Add** > **Empty**. With the Empty selected, go to the Physics buttons, and select **Fields** > **Wind**. Input a non-zero strength, just to see where the wind is pointing. Now rotate the empty so the wind faces the flag. Now it's time to see what Strength is more adequate. In my setup, to get a good soft wind effect, I used a strength of 0.05, if you use something like this you will get results similar to mine. If you set the strength to 1.00 in this setup, you are more likely to get a hurricane.

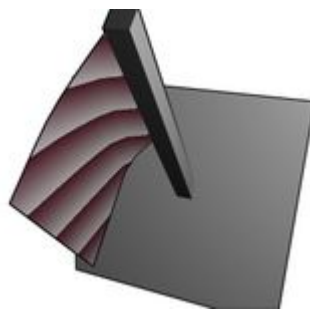


**The wind setup**

To get a nice result, a texture is added, edge rendering (to get a cartoon look), add the pole (a stretched cube) and a wall and you get this cartoon flag. Even though it is a cartoonish look, it is still very realistic in its movement.



Frame 1



Frame 9



Frame 18



Frame 27



Frame 36



Frame 45

## Fine Tuning of Properties and Hints

- Most fiber cloth (cotton, linen, etc) you should consider not having Spring Quads, for a better simulation. For materials that are fiberless, like leather and plastic, enable **Spring Quads**. For most cloth materials, **E Stiff** is greater than 0.975. If you use a value below this, you will get to non-real looking results.
- Decreasing **Error Limit** always leads to better results, but also a significant increase in baking time.
- The procedures used to create our soft block only can be used to create box-shaped objects, to produce curved objects you will have to try other approaches to generate the mesh.
- The smaller the **Interval** when baking, the more realistic movement. Unless you have a strong reason to do otherwise, set it to 1.
- Applying the subdivision modifier is not recommended, instead, subdivide the mesh. While in edit mode (**Tab** key) press **F9** and select **Subdivide** a few times. Each time you subdivide the mesh, the number of its faces is multiplied by four.
- When the Softbody has not been baked yet, the calculation will only be done if you keep going forward in the animation. If you rewind a single frame, it will reset all of the calculation. Also, when you are about to render, bake the simulation, it will improve the render speed a bit.
- The more you subdivide the mesh of the softbody, more accurate your simulation will be.
- The smaller the **E Stiff**, the higher the probability of collapsing if you are working with a

soft solid. A softbody collapsing is not always an error in design. In the real world, if you let an object fall from a great height, it will break too. Depending on the setting (high impact speed, relatively low number of vertices in the impact, aka corner impact) , it will be impossible for the object to not collapse.


- As of now, you cannot simulate the collision of two softbodies, only a softbody VS hardbody collision.
- A value of Stiffness above 0.99 will increase baking time a lot, so try not to put it above this.

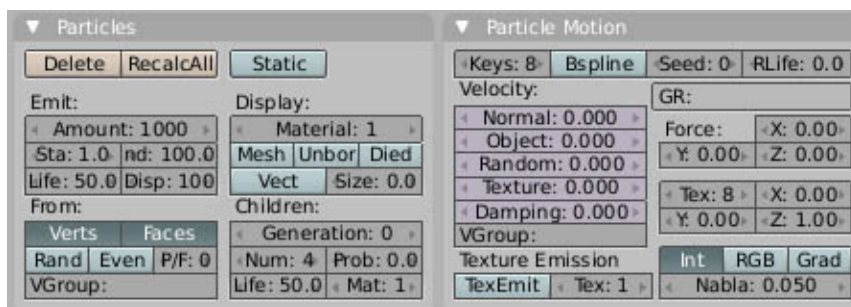
# Particles

## Introduction

Blender has a fast and powerful system to create particles. Particles are the best solution to simulate amorphous effects, like wind, fire, lightning, smoke, etc. Every mesh object can server as an emmitter for particles. Halos can be directly emmitted from particles and using dupliverts, you will be able to emit any type of blender object.

Particles may be set to be deflected (reflected) by other mesh objects, and may be subject to realistic basic forces of nature, like gravity or wind. Also, with the option of static particles, you will be able to generate fur, hair, and grass with a great detail of realism.

To get to it, press **F7**, and  (Physics Button). Click **New** and you will see a screen like the screenshot below, either in two tabs, or two panels like these.



The main particle panel

## Particle Panel

- **Emit:**
  - **Amount:** The total number of particles that will be emmitted.
  - **Sta:** The starting frame of emission.
  - **End:** The last frame of emission.
  - **Life:** How long the particles will exist after emission
  - **Disp:** Percentage of particles displayed and calculated in 3DView.
- **Static:** All the particles are emitted, and are not modified. Combined with vect, can generate "fibers" and is very useful to generate grass, fur, and hair alike.
  - **Animated:** Recalculate Static Particles at each rendered frame, for animation.
- **From:**
  - **Verts:** Vertices of the mesh will be source of particle emission
  - **Faces:** Faces of the mesh will be source of particle emission
  - **Rand:** Random faces will be selected as emitters
  - **Even:** The particles emitted will be proportional to the area of the each face
  - **P/F:** Maximum particles emitted per face. A higher number may generate fuzziness in the emission
  - **VGroup:** If you have created different vertex groups on the object, you will be able to select diferent properties for each one here.
- **Display:**

- **Material:** What material will be used in the particles
- **Mesh:** Also render the emitter mesh
- **Unborn:** Render particles that will be emitted in the future frames
- **Died:** Render particles whose life has ended.
- **Vect:** The particles are enabled to rotate, and they get a local system of coordinates
- **Size:** The distortion caused by the velocity in the shape of the particle
- **Children:**
  - **Generation:** The current generation of particles. You can define properties for up to four generations.
  - **Num:** The number of generations of particles that can multiply itself
  - **Prob:** The probability of a particle of that generation having a child
  - **Life:** The life in frames of the children emitted
  - **Mat:** The material of the children. Using a different materials in the children may be useful, e.g. if you wish to generate smoke from fire.

## Particle motion panel

- **Keys:** How many control points will be calculated for the particle trajectories
  - **Bspline:** Uses a better interpolation between the keys, resulting in a smoother path
  - **Seed:** A number to generate random numbers needed
  - **RLife:** A random life for the particle
- **Velocity:**
  - **Normal:** The particles gain a starting velocity with the direction of their initial normal. For those emitted from a face, it's the face normal, and for those from vertices, it's the vertex normal.
  - **Object:** The particles gain a velocity with a direction relative to the object center
  - **Random:** The particles gain a starting velocity randomly
  - **Texture:** Use a texture to give particles a starting speed.
  - **Damping:** For a non-zero value of damping, the particles suffer a drag force
  - **VGroup:** Here you can define properties for each vertex group defined.
- **Texture Emission:**
  - **TexEmit:** You can use a texture to select the order of emission of the particles in the mesh. The lighter the texture in one point, the sooner the particles near that point will be emitted
  - **Tex:** What texture will be selected for this.
- **GR:** Use this to select a group of field appliers. If none, all will effect.
- **Force:** Makes the particles suffer a force during their lifecycle
- **Tex:** Use a particle to generate a non-uniform force.
- **Int:** Use the light intensity of the texture as a factor to generate the force
  - **RGB:** Uses the RGB values as components of the speed vector
  - **Grab:** Uses the gradient (the ratio of change) of the texture to generate the speed vector. A texture that does not change (uniform color) has a gradient of zero.
  - **Nabla:** The size that will be used for the calculation of the gradient. The smaller, the more precise.

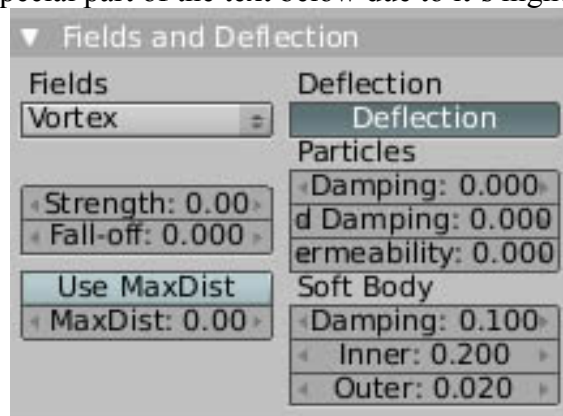
# Particle Interaction

The fields all share the same two properties:

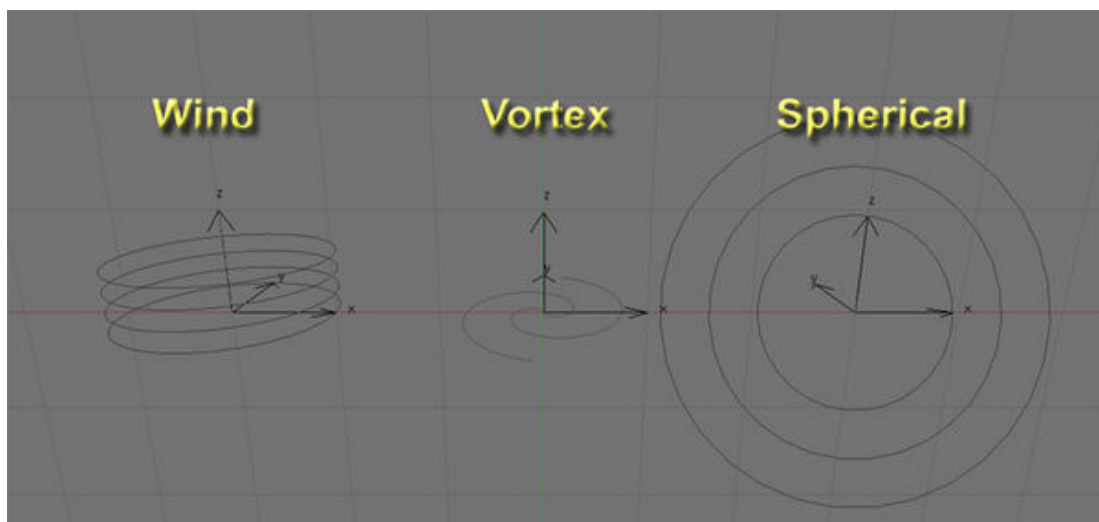
- **Strength:** the strength of the field effect
- **Fall-Off:** how much the distance from the field influences the field strength. A Fall-Off of zero means the the particles are not influenced by the distance to the field applier

There following type of fields :

- Vortex : This is a tornado-like force field
- Spherical : This one either attracts or repels the particles to its center, depending on the strength being positive (attract) or negative (repel)
- Wind : This one makes a force on constant axis for all the particles
- Curve guide : This one attracts or repels the particles based on a curve. This will be discussed in more detail in a special part of the text below due to it's higher complexity.

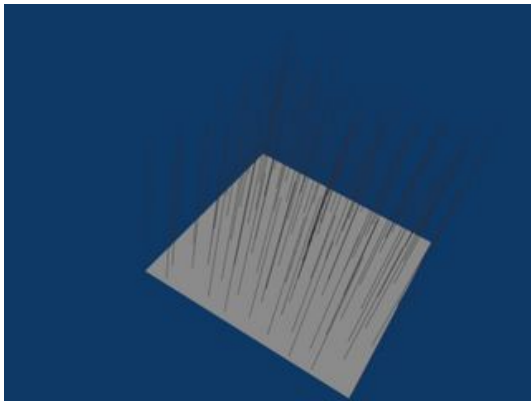


You can use any object as the field applier, but it is a common (and very good practice) to use it in an empty (except for the curve guide), because then the design is cleaner, and the workflow more organized. You can only add one force field per object. If you want more than one force field, you will need more than one object acting as the force field applier.

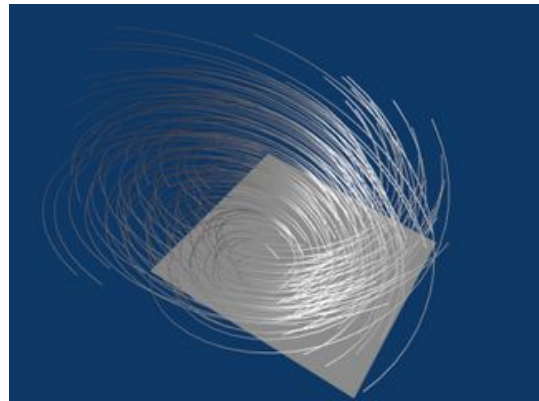


The three basic types of fields, applied on empties

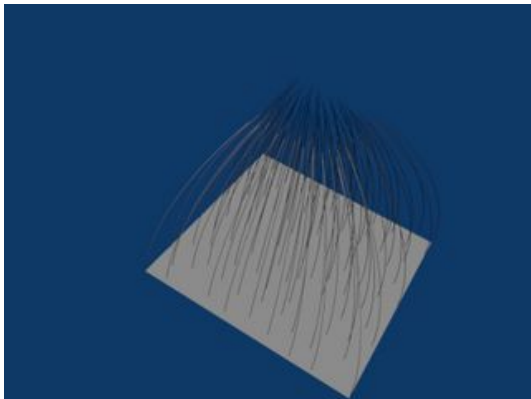




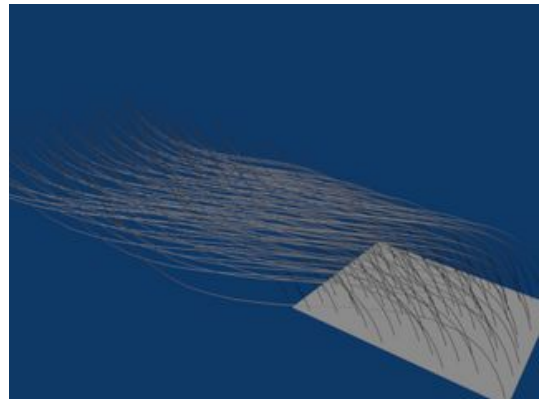
No Fields applied



Vortex applied

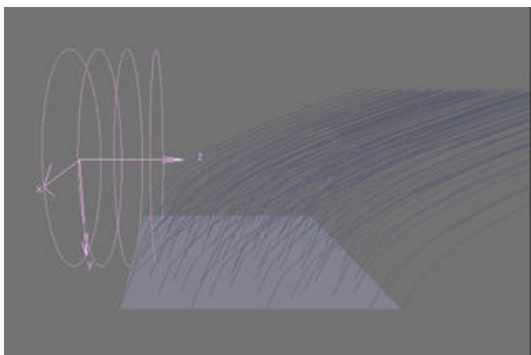


Spherical Field applied

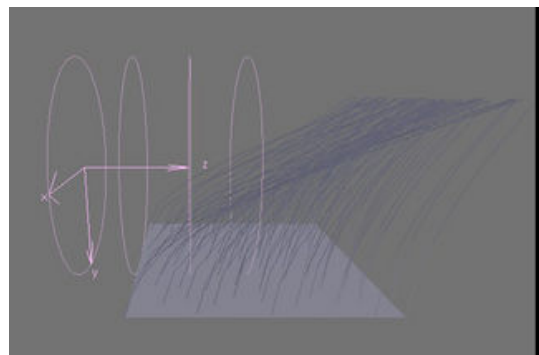


Curve Guide Applied

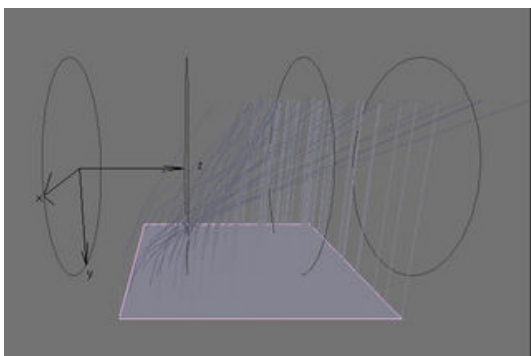
Here you can see the effect of Strength and Fall-off in action using a wind field applied on an empty against a static particle emitter. I suggest you click and enlarge the pictures to get a better view.



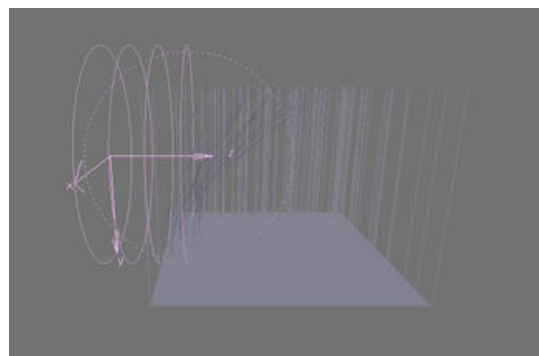
Strength:5,Fall-Off:0



Strength:10,Fall-Off:2



Strength:20,Fall-Off:7



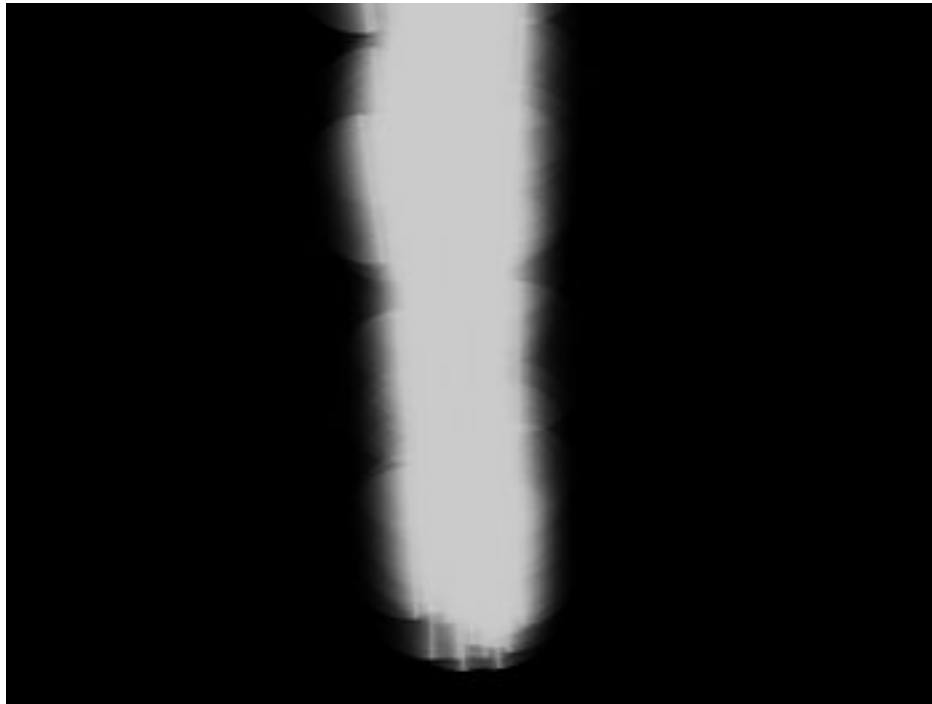
Strength:5,Fall-Off:0,MaxDist=1



## Simulating sparkles

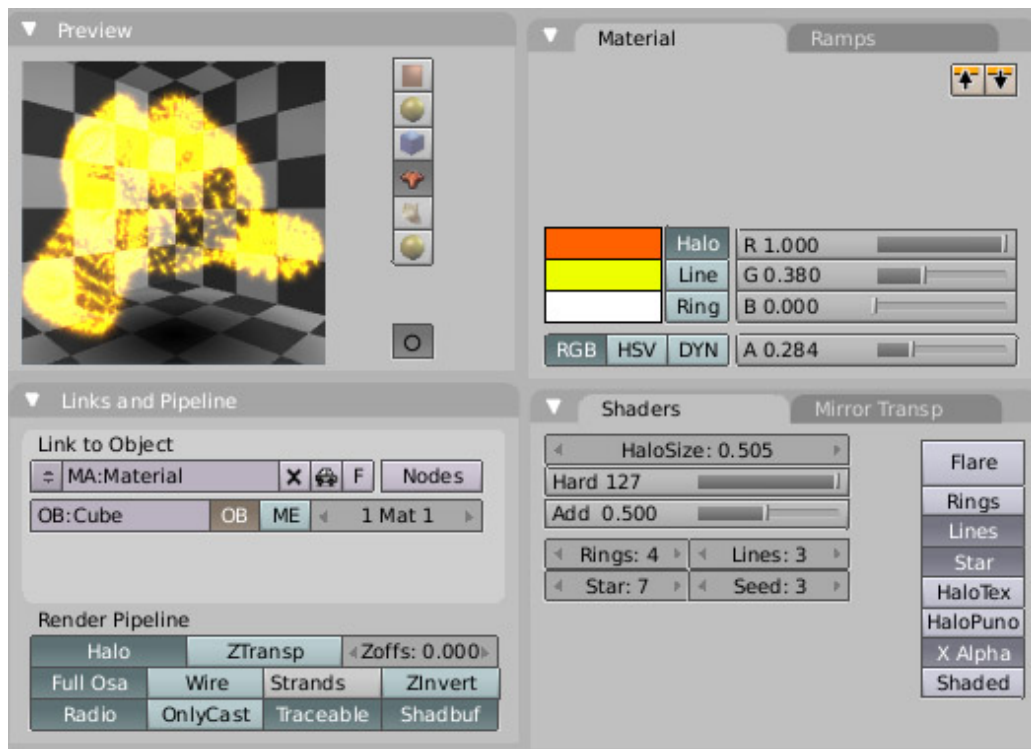
1. Add a plane
2. Scale it down
3. Press **F7** twice and press **New** to enable our plane for particle emission
4. Leave the default settings as they are, just enable **Vect** so our sparkles will stretched by their speed.
5. Go to the particle motion panel/tab, and set **Normal** to 0.070 and **Random** to 0.010 to give them an upwards speed, and a random look. Now, if you advance it some frames, you will be able to see the particles already.

If you render, you will see something like this. Now, let's set a material for our particles.

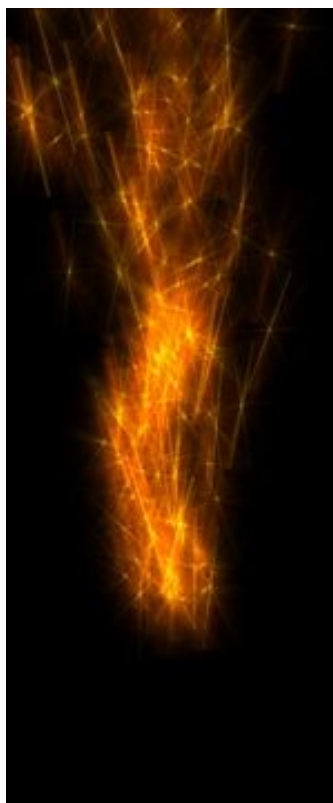


### Our first try

6. Create a material, set it as a **Halo**
7. In the shaders area, select the halo to have **Lines**, **Star**, **X Alpha**, and disable **Flare**, **Rings**, **HaloTex**, **HaloPuno**, **Shaded**. Set **HaloSize** to 0.5, **Hard** to max value, and **Lines** to 3
8. In the material area, select the halo color to be a orange, or red-orange, and the lines to be yellow. Set **Alpha** to a value like 0.2-0.3, so our particles will be semitransparent.
9. Don't get fixed on these values. Tweak them as much as you want and render to see the results. The only thing I recommend is **X Alpha**, because it really increases the definition of our particles.



## The material

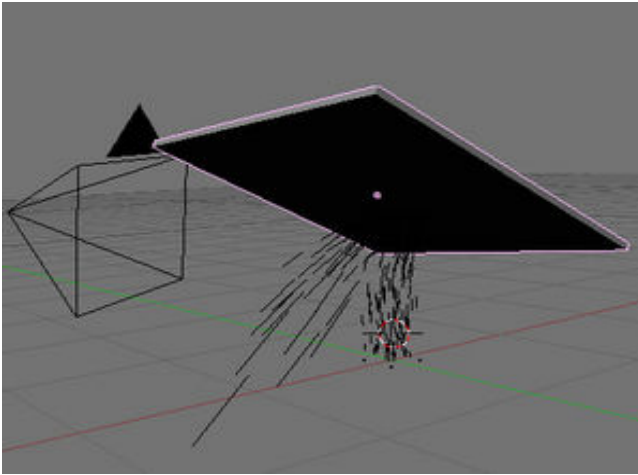


## The final render of our sparkles

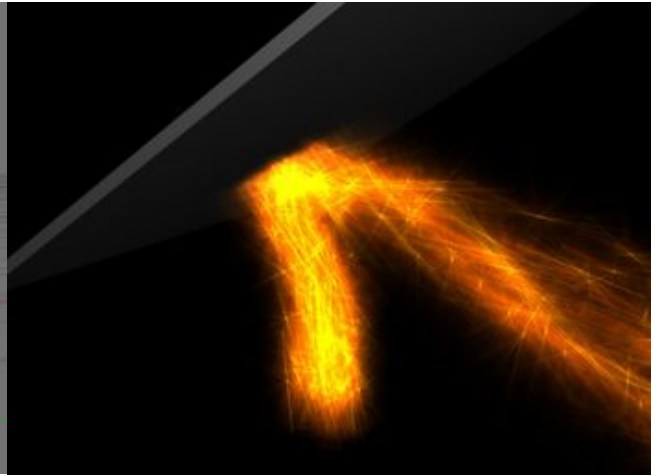
Now a big block will be used to deflect our sparkles.

Add a cube, scale it down in one axis so it becomes thin. Go to the physics buttons (**F7** twice), and enable our block as a deflector. Set its **Permeability** as zero since we don't want any particle to pass through. We don't need to modify **Outer** or **Inner**. By grabbing and rotating it, place our block in a

place it will block particle flow. Give it a slight angle, so the particles deflected will not pass by the same way they came. You probably won't see the results now, since you need to go to our particle emitter and click **Recalc All**. If you have done everything correctly, you will now see the particles bounce off the block.



The scene setup.

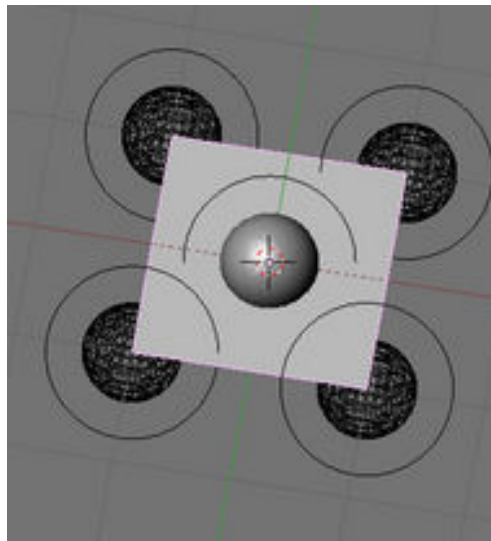


Our final render.

## Emitting objects as particles

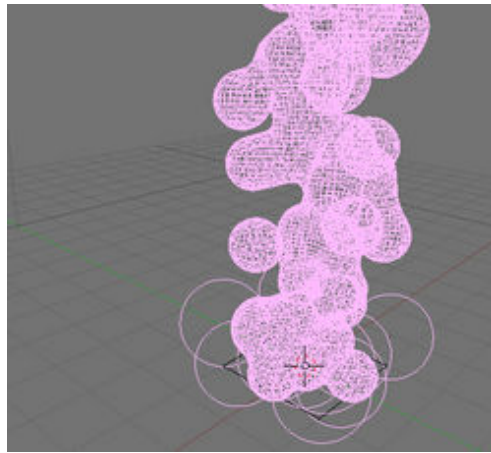
We will now use particles to emit objects. We could choose any blender object, meshes, curves, NURBS, or lamps. But in this case we will use metaballs in order to fake drops of water.

1. Create a plane, and a metaball
2. Parent the metaball to the plane. To do this, select the metaball, while holding **Shift**, select the plane and press **Ctrl+P**
3. Enter object tab **F7**, and select **Dupliverts**. By now you will be seeing our metaballs being emitted



### The dupliverted metaball

4. Enable the plane as a particle emitter. Decrease **Amount** to 100, and give the particles a normal **Speed**.
5. Set a water material with **Raytransp** and **Raymirror**, and you can render.



## The result



## Render

### Good links to follow

Advanced methods in particles: [\[1\]](#)

Creating hair with particles: [\[2\]](#)

Another tutorial on how to create hair with particles: [\[3\]](#)

# Rigid Bodies

## The basics

Sometimes you wish to render a complex scene that involves collisions, multiple forces, friction between multiple bodies, and air drag, but you don't want to try to manually animate each. Luckily, you can count on the Blender game engine to do it for you. Currently, aside from Blender, very few 3d suites offer this function, even professional ones, unless you purchase the most advanced versions. Game engines are designed to simulate rigid (undeformable) bodies. There is currently a plan to integrate this functionality into the Blender's animation system, which will then eliminate the need for baking.

This idealized system can mimic a lot of what happens in our macroscopic world really well and it is this mimicing that is useful for many applications (today's games and animation software included).

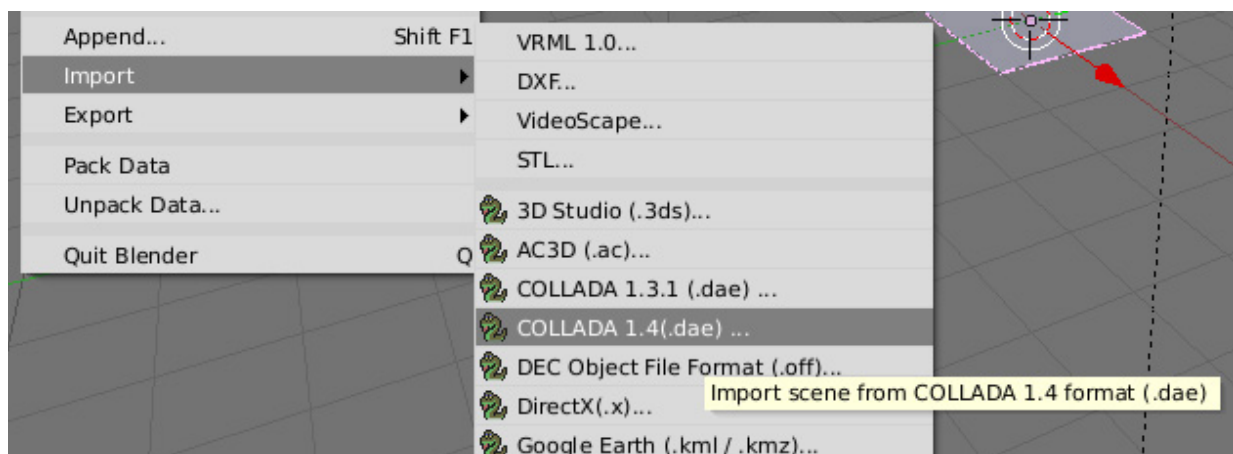
In Blender we can use rigid body functionality to animate the dynamics of objects moving, rotating, and colliding with each other. This is much easier, faster, more realistic, and more fun than trying to recreate it manually. This is the case for even simple cases, not to mention scenes where we can have hundreds or thousands of objects falling and colliding against each other and static obstacles.

## Importing from other programs

In case your 3d suite does not offer this functionality, you can use Blender to generate a rigid body simulation and then export it back to your suite by exporting in the native format of your player from Blender. Though a wiser choice is using the COLLADA format.

COLLADA is a **COLL**aborative **D**esign **A**ctivity for establishing an interchange file format for interactive 3D applications. COLLADA version 1.4, released in January 2006, supports features such as character skinning and morph targets, rigid body dynamics and shader effects for multiple shading languages including the Cg programming language, GLSL and HLSL.

Go to the homepage of the COLLADA project, and search if they currently have a plugin for your 3d suite. If they have it (or you find it on other community driven sites), install it, export from your program, and import in Blender. The guidelines you see will have a set basic method of operation.



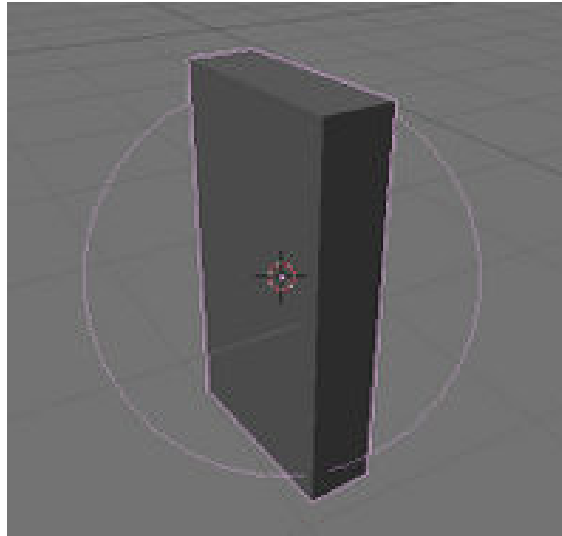
**Importing a COLLADA file from the File > Import menu**

# Dominoes

We will simulate an arrangement of dominoes falling one onto the other. This is a pretty classic simulation for game engines, and very difficult to manually produce, so a good use for IPO Baking.

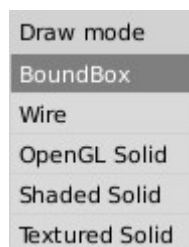
## How to:

*This example does not intend to show how to use the game engine. If you don't know how to use the game engine, however, you will see some very basic guidelines here. The method used here to produce simulations emphasizes accuracy of the simulations, while one designing games wants to ensure a real-time experience. Most of the ways to improve the game engine simulation speed will not be used here, as they may decrease the realism of the simulation.*



Our domino piece, already enabled as actor, as simple as possible.

1. Add a cube, scale it in the axis, so it looks like a domino piece.
2. Enable our cube to be an actor, to be rendered in the game engine. Press F4 and click Actor. Now our domino piece will be emulated by the engine. After selecting our piece as an actor, do not scale it anymore.
3. Select Dynamic,Rigid Body,enable Bounds and select Box on the dropdown menu near Bounds. Dynamic means that our object will respond to gravity, and will move itself when colliding. Rigid body means our object will also rotate when colliding.
4. Create a plane, to be our floor, and move it under the domino piece.

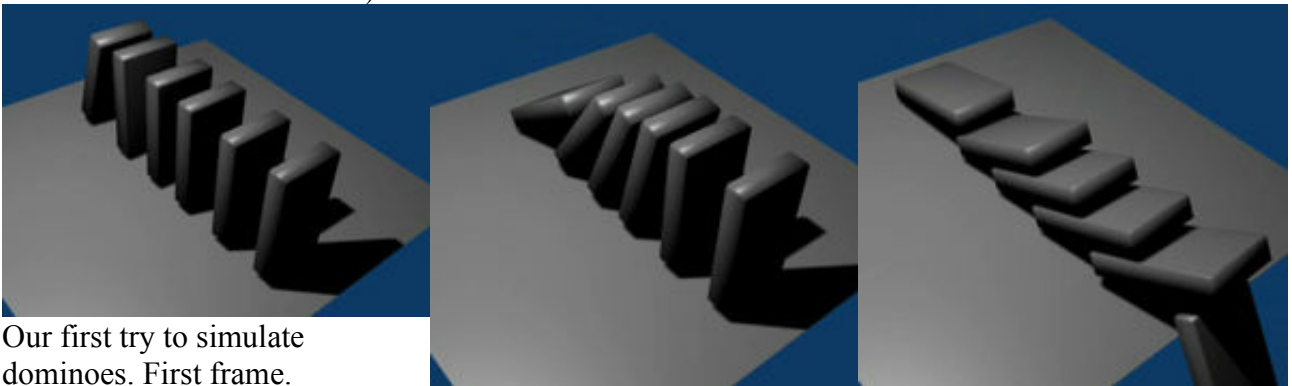


## Selecting the drawmode

5. Now we are ready to duplicate our domino piece and to create a good setup. Use Alt D to duplicate the pieces as linked duplicates. So afterwards, when we edit the mesh of one, all the other will be edited also. To give it our first shot, put six dominoes lined up in a column and rotate the first piece forward a little bit. Press P to run the game engine, and Esc to stop it. It may happen that your first piece doesn't fall on the second, so you will need to rotate it

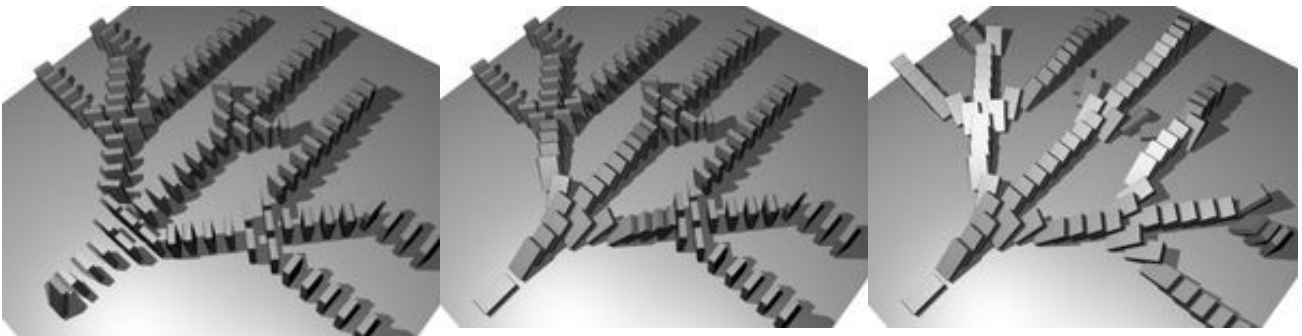
more, just like the example. **Do not enable IPO baking right now.**

6. If your pieces give a little jump when you start. This is because they are too near the plane, move them up a little (or move the plane down a little).
7. If you see all white objects when you play the simulation, change the drawmode. For best viewing the results while simulating, use shaded solid. For best performance, use wire. Press D in the 3D window to get this box of choices.
8. Want to change something, create more pieces? You can do it, and run the game engine again. Satisfied with your simulation? We are now going to bake. Save. In the Game menu (in the top of the screen), select Record Game Physics to IPO and Enable all frames. The recording is near real-time.
9. Now you can render, apply subdivision, apply textures, modify the meshes (as we have used duplicate linked, any modification to a mesh, like a bevel, of a single domino piece will also be done on all others).



Our first try to simulate dominoes. First frame.

If wish to modify you simulation, go to the first frame. Move, rotate, duplicate (do not scale) the pieces as you wish. Now, select all by pressing A, insert a keyframe I, then select LocRot. You can Bake again. The simulation below is an expansion of the first. Now, this simulation renders real time in the game engine, but when you bake it, it takes about ten times more.

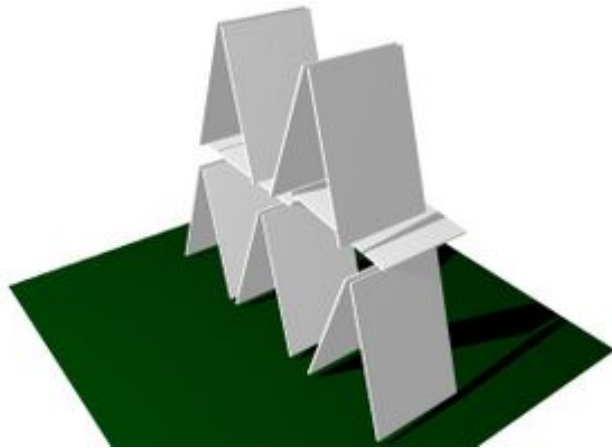


Extension of the first simulation. First frame.

Frame 250.

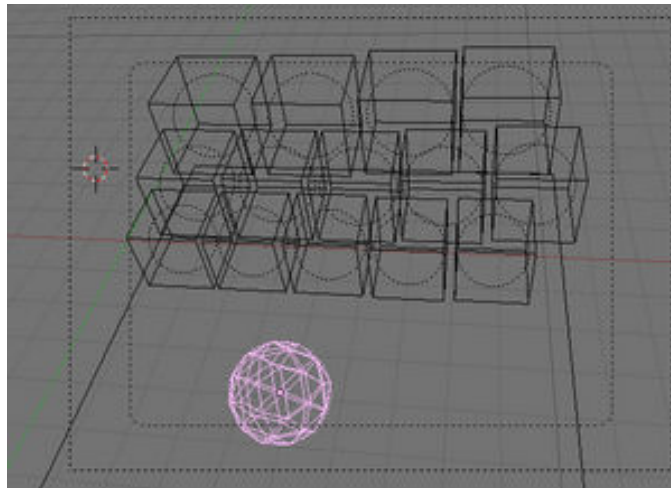
Frame 400.





Another application of the methods seen here: A castle of cards

## Effective use of the game engine



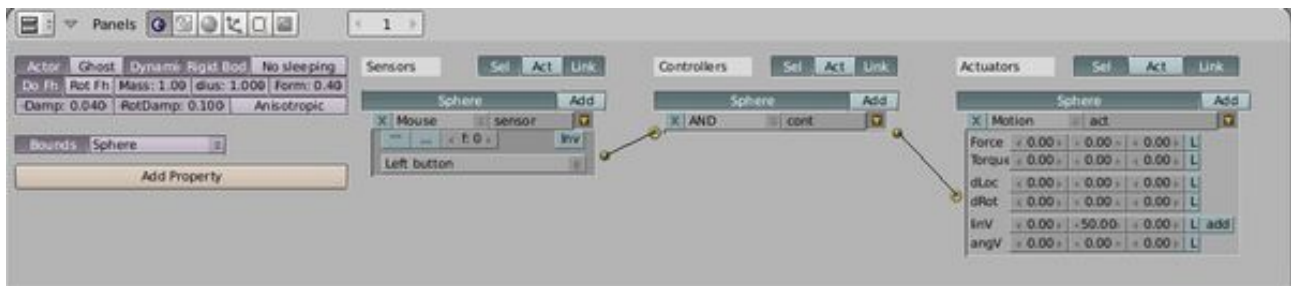
### Our scene setup

Let's suppose you want to interact with the simulation, and to have this baked. We will simulate a game that when you click with the left mouse button, a ball "shoots" against boxes.

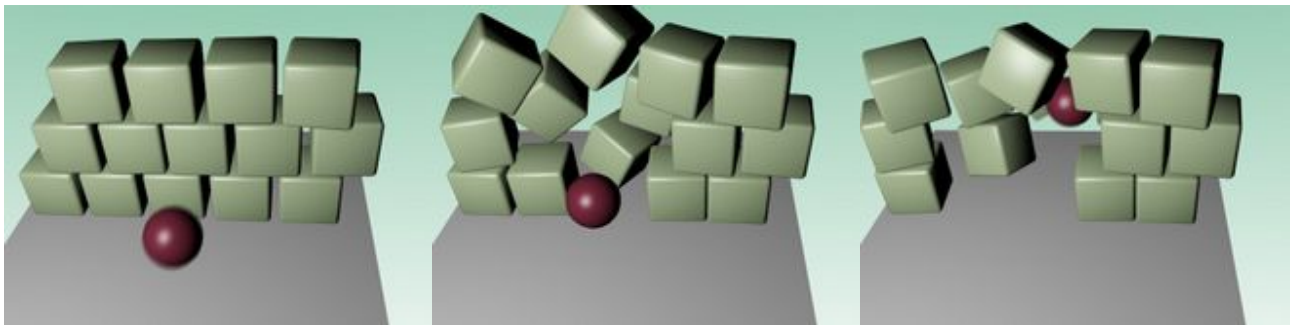
### How to

1. Create a isosphere, and enable it as a actor, enable Dynamic, Rigid Body, and set Bounds to Sphere . To make it move when you press the mouse button, make the setup similar to this: Create a mouse sensor to the left button, an AND controller, and a motion actuator. Set linV in the axis of the collision (in this case, the Y axis) to a high value, like 50, and link them, so it looks like the pic below
2. Create a cube, enable it as an actor, use the same settings used in the domino piece above. Duplicate the cube some more times, and move the copies so it looks like our scene setup
3. Add a plane to be the floor. Now run the game engine, by pressing P.
4. If the ball goes in the wrong direction when you press the mouse button, you have used the wrong axis, or used positive instead of negative orientation.
5. If you liked the results, enable IPO baking, rerun the game engine, and now you can render, apply textures, export to another program, anything.





**Our shooter ball, enabled**



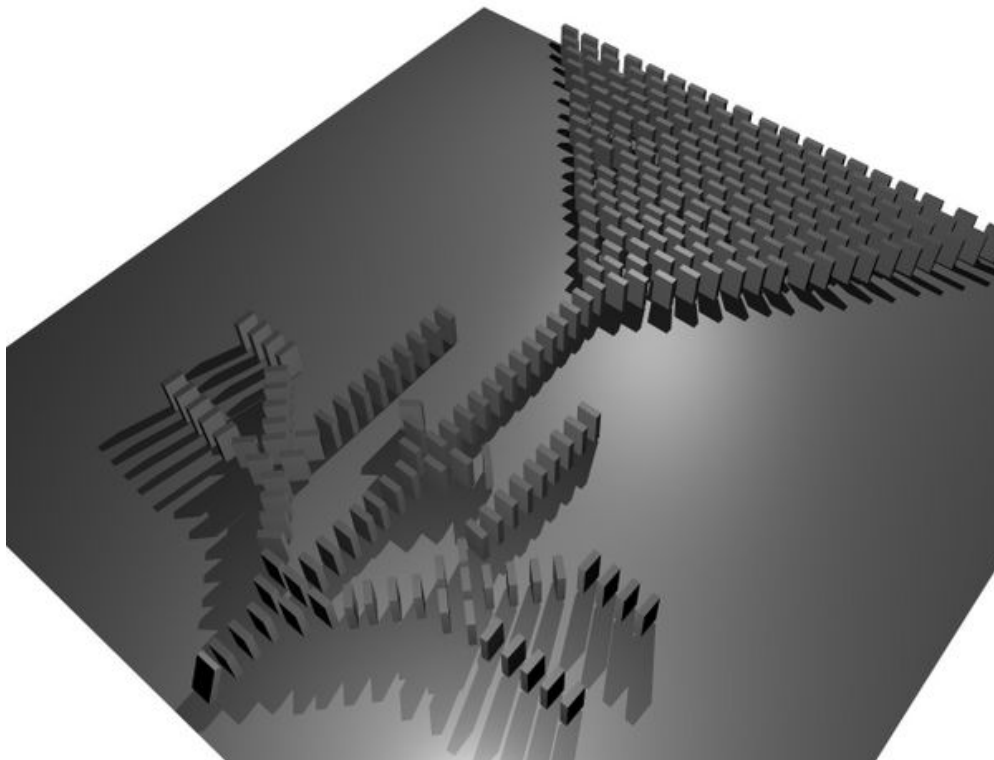
Shooting our ball. First frame, motion blurred

First collision.

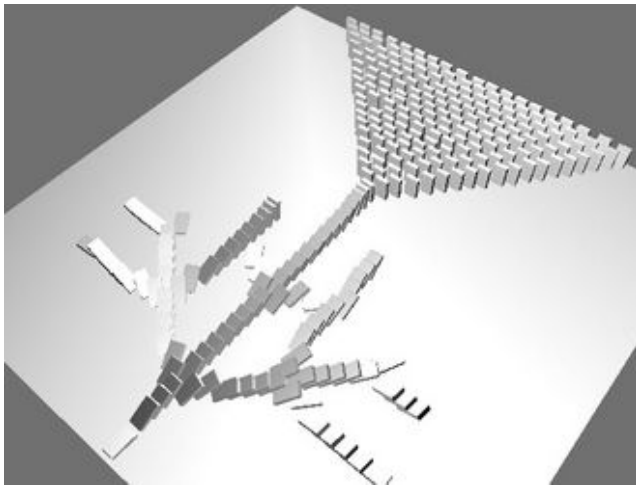
Second collision.

## Huge simulations

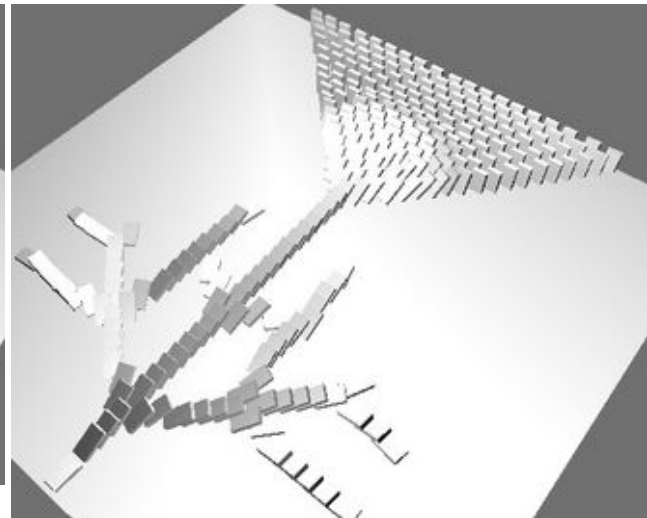
The number of objects is limited by just the amount of RAM memory you have, and by the amount of time you want to wait for the baking. Keep in mind that .blend files with many objects with baked animations can consume gigabytes when saved. Also, a simulation that may be feasible in the game engine becomes much more slow when you enable IPO baking, and can crash the computer.



The first piece is about to fall on the second. This domino set, with 289 pieces, when IPO baked for the first 1000 frames (until all pieces had fallen), and saved, produced a .blend file of more than 2GB. The baking process took about 2 hours, and saving this file took 20 minutes, but just running the simulation without baking was about 40 seconds, until all pieces had fallen. The bounds used was Convex Hull Polytope, for max precision. Using Bounds as box would have decreased simulation time with no sensible decrease in simulation realism.



Baking the above simulation. Frame 300



Frame 500.

## Hints

- From the developers of the Bullet engine, when designing a simulation:
  - Do not scale objects. If you do, apply scale with **Ctrl+A**
  - Keep the masses for the dynamic object similar. If you put an object of 0.1kg resting on an object of 100kg, the results may not be accurate.
  - Assign the right bounds type. For a cylinder, choose cylinder, even for non-moving objects (the same goes for boxes and other shapes). **Convex Hull Polytope** can approximate meshes for moving objects and static objects. **Static triangle mesh** is good for scenarios and terrain.
  - Do not use high values for gravity
  - Do not use too many vertices in objects with **Convex Hull Polytope** enabled.
  - Do not use very large or very small objects (less than 0.1 units).
  - Do not use degenerate triangles in the meshes (triangles where there is one or two very acute angles).
- Objects with higher mass do not fall faster, but require a larger force to be accelerated and stopped. Objects with higher mass only fall faster when there is air drag. You can emulate air drag by using **Damp**. To simulate water environment, use a very high value for damp.
- **Always save before you are going to do IPO Baking.** First, because some bakes can take a long time, and may crash your computer while simulating. After you finished, **save as** to get two files. Second, for big simulations, the size of the files can be very different. Third, version 2.42 has a bug that sometimes does not allow you to erase IPO baked simulations.

## Additional information

COLLADA Homepage: [\[1\]](#)

Developers of COLLADA: [\[2\]](#)

Definition of the COLLADA format: [\[3\]](#)

Blender Collada import/export plugin development page: [\[4\]](#)

Bullet engine homepage: [\[5\]](#)

Development of integration of rigid body simulation inside the Blender Animation system. - [\[6\]](#)